DOI:
10.1109/MC.2014.169

Document Version
Peer reviewed version

Thwarting Obfuscated Malware via Differential Fault Analysis

Guillermo Suarez-Tangil[*]
Department of Computer Science
Universidad Carlos III de Madrid
Madrid, Spain
guillermo.suarez.tangil@uc3m.es

Flavio Lombardi
Consiglio Nazionale delle Ricerche
Rome, Italy
flavio.lombardi@cnr.it

Juan E. Tapiador
Departamento de Informatica
Universidad Carlos III de Madrid
Madrid, Spain
jestevez@inf.uc3m.es

Roberto Di Pietro
Security Activity Research
Bell Labs
Nozay, France
roberto.di_pietro@alcatel-lucent.com

Abstract

Malware for smartphones has rocketed over the last years. As a result, market operators face the challenge of keeping their stores free from malicious apps, a task that has become increasingly complex as malware developers are progressively using advanced techniques to defeat malware detection tools. One such technique, commonly observed in recent malware samples, consists of hiding and obfuscating modules containing malicious functionalities in places that static analysis tools overlook (e.g., within data objects). In this paper, we describe Alterdroid, a dynamic analysis tool for detecting such hidden or obfuscated malware components distributed as parts of an app package. The key idea in Alterdroid consists of analyzing the behavioral differences between the original app and a number of automatically generated versions of it where a number of modifications (*faults*) have been carefully injected. Observable differences in terms of activities that appear or vanish in the modified app are recorded, and this signature is finally analyzed through a pattern-matching process driven by rules that relate different types of hidden functionalities with patterns found in the differential signature. A thorough description of the proposed model is provided. Preliminary experimental results obtained by testing Alterdroid over some of the most relevant malware apps support the quality and viability of our proposal.

Keywords: smartphones, obfuscated malware, grayware, differential fault analysis, fuzzy testing, dynamic analysis, automatic testing, security, privacy

[*] Corresponding author

1 Introduction and Background

Smartphones have rapidly emerged as popular platforms with increasingly powerful computing, communications, and sensing capabilities. Recent commercial surveys show that they are about to outsell the number of PCs worldwide, and that users are already spending as much time on smartphone apps as on the Web [10]. This has caused a substantial rise in the number of apps available for download in online markets, which has contributed to create new business models [5] and reshape the way we communicate, socialize, and access services. Smartphones present a number of security and privacy concerns that are, in many respects, even more alarming than those existing in traditional computing environments [10]. Most smartphone platforms are equipped with multiple sensors that can determine the user location, gestures, moves and other physical activities, as well with cameras and audio-video recording capabilities, to name a few. The sensitive pieces of information that these devices can capture could be leaked by malware residing on the smartphone. Even apparently harmless capabilities have swiftly turned into a potential menace. For example, access to the accelerometer or the gyroscope can be used to infer the location of screen taps and, therefore, to guess what the user is typing (e.g., passwords or message contents) [1]. Similarly, the Radio Data System (RDS) embedded in most AM/FM channels can be exploited to inject attacks on Software Defined Radio (SDR) systems [3].

Most current smartphone platforms are built upon modified versions of desktop operating systems and inherit some traditional security features from them, although they also incorporate more elaborate security models to better fit the architecture and intended use of these devices. One remarkable feature is a permission model aimed at restricting the actions that an app can execute, including access to stored data and other available services. Isolation is generally also enforced through sandboxing and other design measures that regulate inter-application communications. Since a major source of security problems is precisely the ability to incorporate third-party applications from available online markets, security measures at the market level constitute a primary line of defense. Many market operators carry out a revision process over submitted apps that involves some form of security testing. Official details about such revisions remain unknown, but the constant presence of malware in many markets and recent research studies [6] suggest that operators cannot afford to perform an exhaustive analysis over each app submitted for release to the general public. This is further complicated by the fact that determining which applications are malicious and which are not is still a formidable challenge, particularly for the so-called *grayware* –namely, apps that are not fully malicious but that constitute a threat to user security and privacy.

1.1 Malware in Smartphones

The rapid growth of smartphone sales has come hand in hand with a similar increase in the number and sophistication of malicious software targeting these platforms. For example, according to the mobile threat report published by Juniper Networks in 2012, the number of unique malware variants for Android increased by 3,325% during 2011 and by 614% between 2012 and 2013 [7]. Smartphone malware has become a rather profitable business due to the existence of a large number of potential targets and the availability of reuse-oriented malware development methodologies that make exceedingly easy to produce new samples.

Malware analysis is a thriving research area with a substantial amount of still unsolved problems [7]. In the case of smartphones, the impressive growth both in malware and benign apps is making increasingly unaffordable any human-driven analysis of potentially dangerous apps. This has consolidated the need for smart analysis techniques to aid malware analysts in their daily functions. Furthermore, smartphone malware is becoming increasingly stealthy [4] and recent specimens are relying on advanced code obfuscation techniques to evade detection by security analysts [12]. For instance, *DroidKungFu* has been one of the major Android malware

outbreaks. It started on June 2011 and has already spanned over at least six variants. It has been mostly distributed through official or alternative markets by piggybacking the malicious payload into a variety of legitimate applications. Such a payload is encrypted into the app's assets folder and decrypted at runtime using a key placed within a local variable belonging to a specific class module. Another remarkable example is *GingerMaster*, which was the first malware using root exploits for privileged escalation on Android 2.3. The main payload was stored as PNG and JPEG pictures in the asset file, which were then interpreted as code once loaded by a small hook within the app.

More sophisticated obfuscation techniques, particularly in code, are starting to materialize. These techniques and trends create an additional obstacle to malware analysts, who see their task further complicated and have to ultimately rely on carefully controlled dynamic analysis techniques to detect the presence of potentially dangerous pieces of code.

2 Alterdroid: A Differential Fault Analysis Approach

In this paper we describe Alterdroid[1], a tool for detecting obfuscated malware components distributed as parts of an app package. Such components are often hidden outside the app main code components, as these may be subject to static analysis by market operators. The key idea in Alterdroid consists of analyzing the behavioral differences between the original app and an altered version of it where a number of modifications (*faults*) have been carefully introduced. Such modifications are designed to have no observable effect on the app execution, provided that the altered component is actually what it should be and does not have any hidden functionality. For example, replacing the value of some pixels in a picture or a few characters in a string encoding an error message should not affect execution. However, if after doing so it is observed that a dynamic class loading action crashes or a network connection does not take place, it may well be the case that the picture actually contained a piece of code or the string was instead a network address or a URL. Our approach shows some similarities to Fuzzing [9], but our focus is on manipulating components of the program rather than its inputs.

At high level, Alterdroid has two differentiated major components: fault injection and differential analysis (Fig. 1). The first one takes a candidate app –the entire package– as input and generates a fault-injected one. This is done by first extracting all components in the app and then identifying the ones suspected of containing obfuscated functionalities. Such identification is done on an anomaly-detection basis by comparing certain statistical features of the component contents with a predefined model for each possible type of resource (i.e., code, pictures, video, text files, databases, etc.). Faults are then injected into candidate components, which are subsequently repackaged together with the unaltered ones into a new app. This process admits simultaneous injection of different faults into different components and is driven by a search algorithm that attempts to identify where the obfuscated functionalities are hidden. Both the original and the fault-injected apps are then executed under identical conditions (i.e., context and user inputs), and their behavior is monitored and recorded producing two activity signatures. Such signatures are sequential traces of the activities executed by the app, such as opening a network connection, sending or receiving data, loading a dynamic component, sending an SMS, interacting with the file system, etc. Both signatures are then treated as in a string-to-string correction problem, in such a way that computing the Levenshtein (edit) distance between them returns the differential signature, this being a list of observable differences in terms of insertions, deletions, and substitutions. The differential signature is finally analyzed through a pattern-matching process driven by rules that relate different types of hidden functionalities with elements found in the differential signature.

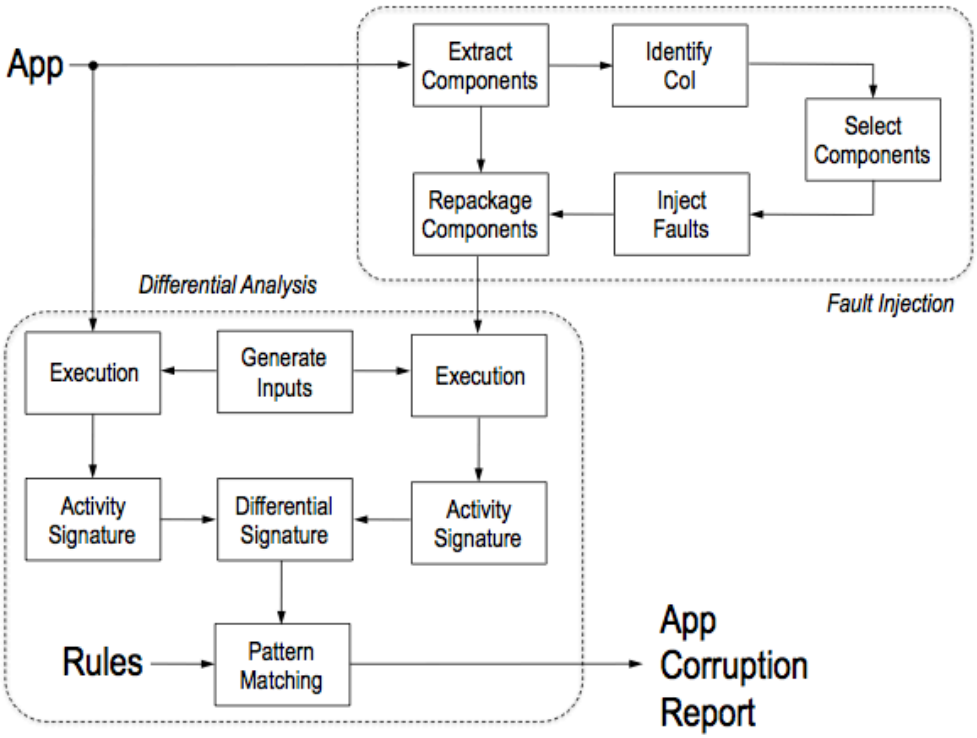

Figure 1: Alterdroid architecture

2.1 Identifying Components of Interest

An app can be seen as a collection of components, each of them composed of a number of classes (i.e., code), as well as by other resources that are accessed at run-time, such as asset files. Components have a type, including code, picture, video, database, etc.. We say that a component is of interest if it does not fit a model defined for all components of its type. In our current version of Alterdroid, models measure statistical features, such as for example the expected entropy, the byte distribution, or the average size. Additionally, we also apply several heuristics based on the file type (as manifested by the *magic number*) and its extension. Such features are computed from a dataset of components of the same type, such as text files, pictures, code, etc. For each model there exists a function that checks whether the component complies with its model or not. For example, if the model is just a byte distribution, then such a function could be a goodness-of-fit test (e.g., chi-squared) between the model and the component's byte distribution. In our experience, such simple models suffice to spot the most common –and rather simple– obfuscation methods observed in smartphone malware, including code camouflaged as supplementary multimedia files, connection data hidden in text variables, etc..

2.2 Generating Fault-injected Apps

Fault conditions can be injected into an app by altering one or more of its components. A Fault Injection Operator (FIO) creates a modified app where a component has been replaced by a slightly changed one. Replacing a component may, or may not, translate into observable differences in the execution of the app. FIOs that replace data components are particularly interesting, as obfuscated malware is especially hard to discover when it is hidden inside what seems to be just data. Nonetheless, FIOs can also be applied to benign components to better understand their role within the app. Some generic FIOs treat components as a bit string and apply changes such as flipping a randomly chosen bit or replacing a portion of the component by another randomly chosen one —chosen among the app's components. In addition, datatype-specific operators can be useful to modify data objects (e.g., multimedia files) while preserving a correct syntax when the focus is on changing the content without rendering the object unusable.

Components of interest identified in the previous stage are injected with faults and reassembled, together with the remaining app components, to generate a mutated app. This process can generate several fault-injected apps, as there are multiple ways of applying different FIOs to different components in the set of Components of Interest (CoI). In Alterdroid, fault-injected apps are generated one at a time and sent for differential analysis. If no evidence of malicious behavior is found in the differential analysis, the fault injection process is invoked again to generate a different mutated app, and so on.

All FIOs in Alterdroid are indistinguishable, meaning that the fault-injected app should behave as the original one, provided that the altered component does not include any hidden functionality. This allows for a more efficient fault injection process based on the fact that the composition of indistinguishable FIOs is an indistinguishable FIO. Consequently, if the same FIO is applied to multiple components and there is a hidden functionality in just one of them, the resulting app will behave exactly as if just the malicious component would have been fault-injected. The overall process, which is entwined with the differential analysis stage discussed later, is essentially a search algorithm that identifies all potentially malicious components.

2.3 Applying Differential Analysis

An app interacts with the platform where it is executed by requesting services through a number of available system calls. We can describe an app's behavior through the *activities* it executes (see also [8]). In some cases there will be a one-to-one correspondence between an activity and a system call, while in others an activity encompasses a sequence of system calls executed in a given order. The execution flow of an app may follow different paths depending on the user-provided inputs and the state of the environment when the execution takes place. In Alterdroid, the observable behavior resulting from the execution is summarized in an *activity signature*, this being an ordered time series given by the sequence of executed activities.

A key task in our system is the analysis of the behavioral differences between the original app and the slightly modified version of it after applying FIOs. This is carried out by recording and analyzing the differences between both apps' observed behaviors, as given by their respective activity signatures. In Alterdroid, we approach this problem as one of string-to-string correction, where differences are represented as the minimum number of edit operations (insertions, deletions and substitutions) that transform one signature into another one. Such a minimal sequence of operations is called the *differential signature*.

The analysis scheme used in Alterdroid is based on deducing properties of an app from the presence or absence of certain *patterns* in the differential signature between the original and the fault-injected app. The next examples illustrate this concept.

Example 1. Let us assume that an app uses an image as icon in its user interface. Modifying some pixels of such an icon, or even replacing it with another valid icon should not affect at all the execution flow of the app. If nonetheless the icon is replaced and the modified app behaves differently from the original app under exactly the same conditions, we can deduce that the original icon contained some unexpected *functionalities*, such as, e.g., a piece of compiled code masqueraded as an icon.

Example 2. Let v be a variable such that its content has no influence on the program flow. For example, v could be a string containing an error message to be displayed at some point. Such strings have been broadly used in existing malware to hide URLs pointing to servers from where the malware can download further code, receive instructions, send data, etc. To avoid detection, the string is often obfuscated, generally through a simple substitution scheme, and the URL is only revealed at execution time. Thus, any modification of the string resulting in a damaged URL will likely prevent the establishment of a connection.

Example 3. Similarly to the examples discussed above, it may be possible to find out whether a component leaks information through a number of sensors (e.g., accelerometer, gps, etc.) if, after fault injection, the differential signature lacks an access to such a sensor and a network connection.

2.4 Implementation

Our prototype implementation of Alterdroid builds on a number of open source Android tools that facilitate tasks such as extracting components, repackaging them back into an app, and analyzing dynamic behavior. Even though in its current version Alterdroid does not have a comprehensive set of fault injection operators and differential analysis rules, the system is easily extensible.

Alterdroid is implemented using generic Java and Python building blocks. App components are extracted using `Androguard`[2]. FIOs are strongly typed during their definition. This is enforced to avoid syntactic or unexpected errors during the execution of the modified app. For instance, an operator defined for a JPG image can only inject faults into components of that type. After a fault is injected, components are repackaged into a modified app using `ApkTool`[3].

Alterdroid uses `Monkey`[4] to generate a common sequence of events to interact with both the original and the fault-injected apps. `Monkey` supports five main classes of input events: *activity launch*, *service launch*, *action buttons*, *screen touch*, and *text input*. Each app is then executed in a controlled environment using the generated stream of events. This functionality is provided by `Droidbox`[5], a sandbox that also allows to monitor various features related to the execution during a fixed, user-given amount of time. In order to generate activity signatures, Alterdroid monitors the execution of 11 different events associated with the app:

- *crypto*: generated when calls to the cryptographic API are invoked.
- *net-open*, *net-read*, *net-write*: associated with network I/O activities –opening a connection, receiving, and sending data, respectively.
- *file-open*, *file-read*, *file-write*: associated with file system I/O activities –opening, reading, and writing a file, respectively.
- *sms*: generated whenever a text message is sent or received.
- *call*: generated whenever a call is done or taken from the device.
- *leak*: generated when a leakage of private information has occurred. This is determined by tainting analysis [2].
- *dexload*: generated when native code is loaded dynamically.

Although our current implementation uses `Droidbox` to monitor the sandbox, our architecture allows the use of other tools for dynamic analysis. For instance, if the target is a malware specialized in escaping from the sandbox, tools based on virtual introspection such as DroidScope [11] might then be used. Furthermore, more efficient leakage detection tools can also be used [7]. Similarly, our architecture is agnostic to the strategy used to explore the app behavior. In our current implementation, our main goal is to intelligently driving the GUI exploration to cover different functionalities. Thus, we do not aim at triggering contextual malware, as its detection is still a formidable challenge [7]. Even though this feature remains out of the scope of our present contribution, we do believe that it will be instrumental in analyzing forthcoming malware and, therefore, will be addressed in future work.

Alterdroid incorporates a pattern-matching engine for analyzing differential signatures as described above. The engine relies on a user-provided set of rules to seek patterns in the differential signatures. Such rules are fully customizable and allow security analysts to express complex properties through classical logical connectors (AND, OR, NOT, etc.).

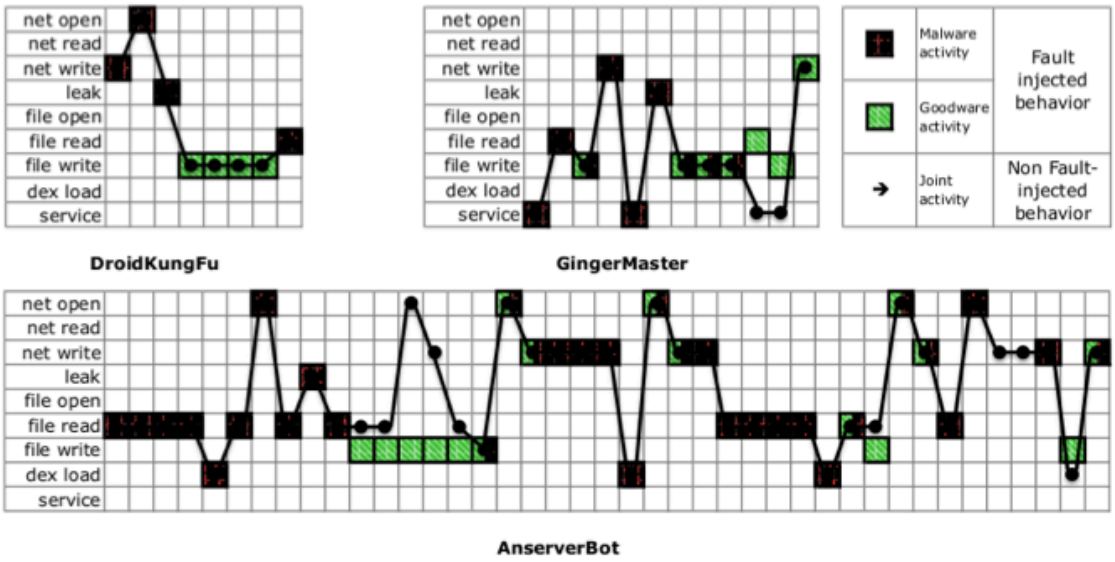

Figure 2: Malware's activity during a time span of 120 seconds.

Finally, Alterdroid allows several parallel executions at the same time. This is due to the amount of time required to execute a dynamic analysis. In fact, the time required to identify the components of interest and generate the FIOs is negligible when compared to the time required to stimulate and monitor the sandbox. We elaborate on performance issues in next section.

3 Alterdroid in Practice: Three Case Studies

We next illustrate how Alterdroid can be used by market operators and security analysts to facilitate the analysis of complex obfuscated mobile malware. We present three case studies of malicious apps found in Android markets: *GingerBread*, *DroidKungFu*, and *AnserverBot*. These three samples constitute representative cases as they incorporate obfuscation techniques of various degrees of sophistication, as well as some malicious features common in malware for smart devices [7] such as aggressive privilege escalation exploits, C&C-like functions and information leakage. Figure 2 summarizes the behavior of the malware that will be discussed throughout this section. Subsequently, we evaluate the performance of our approach over a number of malware samples found in the wild [12].

3.1 DroidKungFu

DroidKungFu (DKF) is one of the major Android malware outbreaks. DKF's main goal is to collect a variety of information on the infected device, including the IMEI number, phone model, as well as the Android OS version.

DKF is mostly distributed through open or alternative markets through repackaging, i.e., by piggybacking the malicious payload into a variety of legitimate applications. Apps infected with DKF are distributed together with a root exploit hidden within the app's assets, namely Rage Against the Cage (RAC). In order to hinder static analysis, this encrypted payload is only decrypted at runtime. In this case study, we analyze one DFK variant by first extracting its components of interest and further applying fault injection and differential analysis over them. We observed that the sample contained about 170 resource files, including PNG (153 files), MP3 (6 files), XML (2 files), DEX (1 file) and RSA key file, among others. All these assets are, in principle, suspected of containing obfuscated functionality. We note here that applying stand-alone static detection techniques would not be enough to identify malicious payloads without requiring human-driven inspections. This is due to the way that DFK obfuscates its core

components. Specifically, each variant uses a different encryption key hidden throughout the code. Even when we attempt to apply stand-alone dynamic analysis, we observe that this technique only gives a rough notion of the holistic behavior of the app. In fact, the behavior introduced by DFK is strongly entwined with the original code of the repackaged app, in such a way that some of its key activities, such as for instance network connections, might be easily seen as normal.

The above-mentioned variant of DFK was fed to Alterdroid. It first identified a number of components of interest, being in all the cases assets associated with the app. Various faults were then injected into such components, and the resulting app was executed and compared with the original one. Figure 2 (DroidKungFu) graphically shows the differential behavior reported by Alterdroid when analyzing such fault-injected app. Activities launched by the original piggybacked app correspond to the full plot, while the behavior after fault injection is given just by the green (legitimate app) and red spots (DKF). In this particular case, a text file pertaining to the assets was randomly modified. This file was later identified as the component containing the RAC exploit. Our analysis shows that disabling the access to such a functionality stops the malware from: establishing a network connection (*net-open*, *net-write*), leaking some information through it (*leak*), and later performing some Input-Output (I/O) operations (*file-read*). These findings agree with previous reports about DKF, including those undertaken by Jiang and Zhou [12].

3.2 AnserverBot

Our second case study deals with *AnserverBot* (ASB), a specimen similar to the first versions of DKF in terms of sophistication and distribution strategy [8]. However, ASB introduces an update component that allows it to retrieve at runtime secondary payloads and the latest C&C URLs from public blogs. Additionally, it also incorporates advanced anti-analysis methods to avoid detection. On the one hand, it introduces an integrity component to check if the app has been modified. On the other hand, it piggybacks the main payload in native runnable code. Furthermore, ASB obfuscates its internal classes and methods, and partitions the main payload in two different parts: while one of them will be installed, the other one is dynamically loaded without actually being installed. More specifically, ASB hides one of these components into the assets folder under any of the following names: *anservera.db* or *anserverb.db*. Furthermore, ASB inserts a new component named *com.sec.android.provider.drm* that executes a root exploit known as *Asroot* [4].

As in the case of DFK, we observed that all ASB samples contain a non-negligible amount of candidate components to be analyzed. The specimen we deal with in this case study contained about 78 resource files, including 54 image files, one database, one DEX file, and a ZIP file, to name a few. After a few iterations of the fault injection process Alterdroid succeeds in positively identifying the actual payload within the DB file, as well as the behavior related to such component. More precisely, this CoI is triggered after observing a mismatch between the magic number of the file (APK) and the actual extension of the database (DB). In fact, when a fault is injected over the database, the ASB's integrity check naturally aborts its execution and produces a result similar to that expected from the original app. Figure 2 (AnserverBot) graphically shows the exhibited differential behavior. As observed, ASB first establishes a network connection (*net-open* and *net-write*) after loading the main payload (*file-read* operations followed by *dexload*). After that, it keeps on reading data that is finally leaked out. Interestingly, the legitimate application uses the network as well, although it does not leak any personal information.

3.3 GingerMaster

GingerMaster (GM) was the first known Android malware to use root exploits for privilege escalation on Android 2.3. GM's main goal is to exfiltrate private information such as the device

ID (IMEI, MSI, etc.) or the contact list stored in the phone. GM is generally repackaged with a root exploit known as *GingerBreak* [4], which is stored as a PNG and a JPG asset file. Right after infecting the device, GM connects to the C&C server and fetches new payloads.

We analyzed a GM sample containing around 61 asset resources, 30 of which were pictures in different formats. From those 30 pictures, Alterdroid identified 4 as strongly suspicious. (Actually, a detailed analysis shows that they are malformed PNGs and that they also contain several ASCII text scripts.) Alterdroid was also able to identify that such malformed images files play a key role in triggering the payloads piggybacked into the legitimate app, including the ASCII text scripts.

Figure 2 (GingerMaster) shows the differential behavior obtained when one of such images is fault injected. Our analysis shows that GM starts the execution of a *service* that performs some IO operations (*file-read* and *file-write*) before finally leaking private information through the network (*net-write* and *leak*). Again, even when the malicious components are hidden, Alterdroid proved to be able to discriminate them and facilitate the identification of the underlying malicious behavior.

3.4 Evaluation

We tested Alterdroid against a dataset composed of around 300 obfuscated malicious apps, grayware [7], and goodware. More precisely, we tested about 200 variants of DKF, ASB, and GM obtained from the Android Malware Genome Project [12] and about 100 legitimate apps available from the RAMP Competition[6]. Every app was executed over a time span of 120 seconds, except for the GM+ ones, which required 1,200 seconds. Table 1 summarizes the experimental results obtained and shows both the performance of the analysis and effectiveness of the detection (full reports and details about the dataset are available online[1]). For the analysis part, performance is measured as the time (overhead) taken by Alterdroid per app and FIO, including the duration of the dynamic analysis. Detection effectiveness is given by the True Positive (TP) rate; that is, the ratio between the number of correct positive matches and the total number of app tested. False Negatives (FN) correspond to apps that are supposed to match a rule, but Alterdroid fails to do so. As for legitimate apps, during our evaluation we didn't inspect every suspicious behavior reported for them. Instead, those apps that are presumably good but that matched one or more rules were labeled as grayware and marked for subsequent analysis. Recall that Alterdroid is more an analysis than a detection tool, as the chief goal is identifying obfuscated functionality and reporting it to the analyst. However, classifying hidden functionality as malicious or not is an entirely different problem that, in many cases, may depend on each user's privacy preferences.

	Apps	CoIs	FIOs	Matches	Overhead	TPs	FNs	Accuracy
DKF	34	6.11	6.11	11.71	283.93 s	33	1	97.06%
ASB	187	1.35	1.35	3.90	246.22 s	186	1	99.47%
GM	4	4.00	4.00	6.00	248.23 s	3	1	75%
GM+	4	4.00	4.00	3.00	1026.01 s	4	0	100%
Grayware	16	2.88	2.88	4.19	248.24 s	16	0	100%
Goodware	81	0.57	0.57	0.00	0.00 s	81	0	100%

Table 1: *Evaluation against existing malware, grayware, and goodware apps. The number of CoIs, FIOs, Matches, is given on average per app, and the overhead is given on average per FIO and app.*

One interesting aspect of Alterdroid is that the dynamic analysis process accounts for approximately 50% of the total time overhead. The remaining 50% is taken by `Droidbox` to set up and boot the sandbox. For example, analyzing the CoIs of an app and injecting 4 faults takes

about 10 seconds, whereas starting up an Android 2.3 sandbox with 512 MB of RAM takes about 160 seconds per FIO. These figures reveal that the performance of the entire differential analysis process is strongly affected by the preparations required for the dynamic analysis. This suggests a simple optimization strategy based on booting a pool of sandboxes and keeping them alive, rather than initializing them on demand.

Selecting an optimal value for the duration that suffices to extract a complete differential signature highly depends on the type of malware and on the mechanisms it uses to trigger the malicious payload, as seen for example in GM and GM+. However, we noticed that most specimens are quite eager to run their payloads promptly. Thus, monitoring only a short period of time is generally enough to extract a reliable evaluation.

As for detection accuracy, our experiments show that Alterdroid performs very well, especially when dealing with obfuscated malware (DKF, ASB and GM). The only case where accuracy drops below 97% (GM) was related to the short time given to dynamic analysis. This is corrected (GM+) by just increasing it. For instance, results on ASB shows an average number of 3.90 rules matching several differential signatures with suspicious behaviors, such as network or data leakage activity.

One challenge we faced when analyzing legitimate apps classified as grayware was identifying whether some behaviors were malicious or not. Many legitimate apps are not fully malicious but carry out activities that may constitute a privacy risk for some users. During our analysis, we detected that most of such suspicious behaviors are related with accessing local data and exfiltrating it over the network. We did not analyze in detail whether this was an intrinsic behavior of the app caused by the fault-injection process, for example because the app contained an integrity check similar to the one discussed for ASB. Nonetheless, this indicates that the app is behaving suspiciously and therefore it could be worth further analyzing it.

4 Conclusions

Today's mobile security requires new approaches to protect users' devices since traditional detection techniques are overwhelmed by the sophistication and obfuscation of current mobile malware [5]. Furthermore, the current state of the art and trends in mobile malware suggest that automated malware detection and analysis is a major requirement for apps review.

Differential fault analysis, as implemented by Alterdroid, is a powerful and novel dynamic analysis technique that can identify potentially malicious components hidden within an app package. Alterdroid is a good complement to static analysis tools, that are focused on inspecting code components, hence possibly overlooking pieces of code hidden in data objects or just obfuscated.

Even though Alterdroid is a first proof of concept, it has been conceived as a general-purpose framework with a very versatile architecture that can be extended in a number of ways. We are currently working on an open-source engineered version of Alterdroid striving on offering an automatic tool for malware analysis. Furthermore, based on our preliminary results, we show that Alterdroid's effectiveness does not directly depend on the duration of the dynamic analysis, but rather on how "fast" the malicious conditions are triggered. Although current malware is extremely naïve in this regard, we believe that future specimens will be more resilient against this detection technique, hence calling for further efforts in contextual security [7]. Similarly, one mayor challenge in fuzzing is to obtain an effective testing by appropriately selecting the components of interest and finding an optimal way to inject faults [9]. Thus, further efforts in this direction should also be addressed.

References

[1] L. Cai and H. Chen. "Touchlogger: inferring keystrokes on touch screen from smartphone motion". In *Proceedings of the 6th USENIX conference on Hot topics in security*, HotSec'11, pages 9–9, Berkeley, CA, USA, 2011.

[2] W. Enck, P. Gilbert, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[3] E. Fernandes, B. Crispo, and M. Conti. "Fm 99.9, radio virus: Exploiting fm radio broadcasts for malware deployment". *IEEE Transactions on Information Forensics and Security*, 2013.

[4] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. "Riskranker: scalable and accurate zero-day Android malware detection". In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[5] N. Leavitt. "Today's mobile security requires a new approach". *IEEE Computer*, 46(11):16–19, 2013.

[6] J. Oberheide and C. Miller. "Dissecting the android bouncer". *SummerCon2012, New York*, 2012.

[7] G. Suarez-Tangil, J.E. Tapiador, P. Peris-Lopez, and A. Ribagorda. "Evolution, detection and analysis of malware for smart devices". *IEEE Communications Surveys & Tutorials*, November 2013.

[8] G. Suarez-Tangil, J.E. Tapiador, P. Peris-Lopez, and J. Blasco. "Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families". *Expert Systems with Applications*, 41(1):1104–1117, 2014.

[9] A. Takanen, J.D Demott, and C. Miller. "*Fuzzing for software security testing and quality assurance"*. Artech House, 2008.

[10] Y. Wang, K. Streff, and S. Raman. "Smartphone security challenges". *IEEE Computer*, 45(12):52–58, 2012.

[11] L.K. Yan and H. Yin. "Droidscope: seamlessly reconstructing the os and Dalvik semantic views for dynamic Android malware analysis". In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[12] Y. Zhou and X. Jiang. "Dissecting Android malware: Characterization and evolution". In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

Footnotes

[1]Available online at: http://www.seg.inf.uc3m.es/~guillermo-suarez-tangil/Alterdroid/

[2]AndroGuard. https://code.google.com/p/androguard/

[3]ApkTool. https://code.google.com/p/android-apktool/

[4]Android Developers. http://developer.android.com/

[5]DroidBox. https://code.google.com/p/droidbox

[6]RAMP Competition. http://fsktm.upm.edu.my/~alid/RAMP.htm

Biography

Guillermo Suarez-Tangil is a PhD student in the Computer Security (COSEC) Lab at Universidad Carlos III de Madrid, Spain. His research focuses on security in smart devices, intrusion detection, event correlation, and cyber security. He has participated in various research projects related to network security and trusted computing. He holds a B.Sc. and a M.Sc. in Computer Science from Universidad Carlos III de Madrid.

Flavio Lombardi, researcher at National Research Council and Adjunct Professor of Computer Science at Dept. of Mathematics and Physics of Roma Tre University. He is member of the Security and PRivacy INnovation GRoup (SPRINGER) at Roma Tre. He received a Ph.D. in Computer Science from Dept. of Computer Science at Sapienza, University of Rome. His main research interests are in the areas of: Cloud and Virtualization security; GPGPU computing; security and privacy for mobile and distributed systems.

Juan E. Tapiador is Associate Professor of Computer Science in the Computer Security (COSEC) Lab at Universidad Carlos III de Madrid, Spain. Prior to joining UC3M, he was Research Associate at the University of York, UK. His work back there was funded by the ITA project (www.usukita.org), a joint effort between the UK Ministry of Defence and the US Army Research Lab led by IBM. His main research interests are in computer/network security and applied cryptography. He holds a M.Sc. in Computer Science from the University of Granada (2000), where he obtained the Best Student Academic Award, and a Ph.D. in Computer Science (2004) from the same university.

Roberto Di Pietro is with Bell Labs, Security Activity research. His main research interests include: security and privacy for wireless systems; cloud and virtualization security; security and privacy for distributed systems; applied cryptography; computer forensics, and role mining for access control systems (RBAC).